A Virtual-Channel Free Mapping for Application-Specific On-Chip Torus Networks

Hiroki Matsutani
Keio University
3-14-1 Hiyoshi, Kohoku-ku,
Yokohama, JAPAN

Michihiro Koibuchi
National Institute of Informatics
2-1-2 Hitotsubashi,
Chiyoda-ku, Tokyo, JAPAN

Hideharu Amano
Keio University
3-14-1 Hiyoshi,Kohoku-ku,
Yokohama, JAPAN

Abstract

Various types of Networks-on-Chips (NoCs) have been employed light-weight routers compared with those in parallel computers, and a virtual-channel mechanism, which requires additional logic and pipeline stages, is one of the crucial factors for a low cost implementation of an NoC router in the case of simple tile-based architectures. Although a torus network, which exploits wrap-around channels, achieves higher throughput and lower latency than a same-sized mesh, a virtual-channel mechanism is usually required to avoid deadlocks in tori with dimension-order routing. In this paper, we propose a scheme to remove virtual channels in tori by accomplishing the following steps: 1) providing a mechanism which allows wrap-around channels to be individually disabled in each router, 2) a task mapping strategy that carefully assigns tasks to a tori, so that as many wrap-around channels as possible are exploited without introducing deadlocks or performance degradation. Additionally, we extend this strategy to avoid deadlocks when application traffic patterns are unknown or incompletely analyzed. Although the proposed mapping does not use virtual channels, it achieves almost the same performance as conventional mapping in tori in ten traces. Moreover, the hardware amount of the proposed router can be decreased to 52.4% of a conventional router providing two virtual channels for tori.

1 Introduction

Various types of Networks-on-Chips (NoCs) have been studied to connect a number of modules in a chip by introducing a network structure similar to that in parallel computers[1, 3]. They are able to employ a complicated network protocol stack, and they are usually suitable for connecting a number of modules.

One of the major target applications of NoCs is stream processing such as JPEG or MPEG coders mostly used in consumer equipments. Figure 1 shows a task diagram of JPEG2000 decoding. In the processing, each task can be mapped onto each node [1] , and is performed in a pipelined manner. In this case, the communication is limited only between neighboring two nodes. However, the framed part called EBCOT (Embedded Block Coding with Optimal Truncation) requires a large computational load and bottlenecks the whole stream flow if each processing in EBCOT is assigned into a single node. For equalizing the stream flow, the processing of EBCOT should be distributed into several nodes and executed in parallel. In the parallelized case, the communication pattern between nodes includes a stream fork and join.

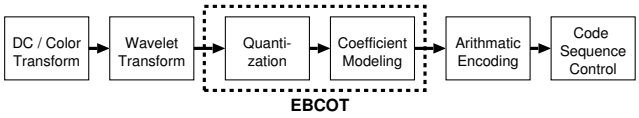

Figure 1: A task graph for JPEG2000 decoding.

Two-dimensional mesh [2, 6, 9] and torus[3, 7] have been employed as the NoC topology, because only a small number of ports (up to four) are required to connect with the neighboring routers. A simple and popular deterministic routing on such networks is dimension-order routing, which simply sends packets on Y direction after completing on X direction. In a router, its routing algorithm has been implemented using a small-dedicated logic, and a routing table that requires register files for storing routing paths is hardly used. Dimension-order routing is simple but it can uniformly distribute minimal paths between all pairs of nodes in the case of uniform traffic.

Although a torus network has twice bisection bandwidth of a same-sized mesh, dimension-order routing originally requires two virtual channels to avoid deadlocks in the case of tori. Whereas a mesh network requires no virtual channels.

A virtual-channel mechanism strongly affects the router architecture, its hardware amount, and latency

[1] In this paper, we use the term "node" to represent the tile where a set of an IP core and a router is included.

in the case of the simple tile-based architectures. Although an input buffer is a crucial component in simple NoC routers, a separate buffer memory is required for each virtual channel in order to switch flits from multiple virtual channels to the same physical channel in a single cycle[4]. Typical router architectures are pipelined at the flit level, and an additional pipeline stage is usually required for the virtual-channel mechanism so as not to pack more logic into a stage. For example, each header flit travels through the steps of routing computation, virtual-channel allocation, crossbar allocation, and crossbar traversal[4]. The pipeline depth is directly related to the depth of the input buffers and latency of the networks. Thus, a virtual-channel mechanism is one of the crucial factors for low cost implementations of simple NoC wormhole routers.

In recent design methodologies, embedded systems and their applications are designed with system level description languages like System-C, and simulated in the early stage of design in the simple tile-based architectures. By analyzing the communication pattern between computational tasks in the target application, we can statically optimize its task mapping[6]. To remove virtual channels from dimension-ordered routers on tori, we propose a task mapping strategy that carefully assigns tasks onto a torus so that as many wrap-around channels as possible are exploited without introducing deadlocks or performance degradation. Additionally, we extend this strategy to avoid deadlocks when application traffic patterns are unknown or incompletely analyzed.

Notice that virtual channels play a key role to avoid not only deadlocks, but also head-of-line blocking. However, average hop count is small in the cases of stream processing, so the advantage of increasing throughput is not fully extracted in our target NoCs.

The rest of this paper is organized as follows. Existing virtual-channel free solutions to route packets are surveyed in Section 2, and our task mapping strategy is proposed in Section 3. In Section 4, the mapping strategy was applied to real application traces and the results are compared with a virtual-channel router on tori. In addition, hardware amount of the proposed router is compared with that of a conventional virtual-channel router. Finally, we conclude in Section 5.

2 Related Work

To remove virtual channels from dimension-ordered routers on tori, the bubble flow control[8] has been developed mainly for parallel computers. This is an injection limitation mechanism that guarantees continuous message movement in the network by preserving at least one empty packet in routers' queue.

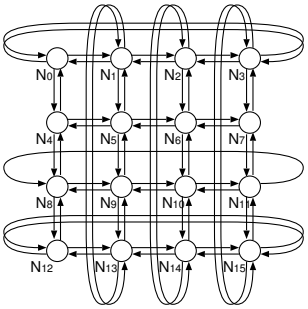

Figure 2: An example of 2-D reconfigurable torus.

To preserve buffers for one or more packets, the bubble flow control is used with virtual-cut through (VCT) switching that requires buffers with capacities for one or more whole packets. However, since the buffer size is a crucial factor for implementing lightweight NoCs described in Section 1, wormhole switching technique has been employed in the case of NoCs.

Another technique to remove cyclic dependencies is based on temporally storing packets that could introduce channel cyclic dependencies at an intermediate node[5]. It is mainly intended to the case of system area networks (SANs) with no virtual channels, such as Myrinet. This technique is used to efficiently implement various routing algorithms on SANs. On the other hand, NoCs (intra-chip communication) are quite latency-sensitive compared with SANs (inter-chip). The latency to store and reinject packets on an intermediate node cannot be ignored on NoCs, and the large number of store-and-reinjected packets introduce the serious performance degradation. Thus, this method would not be directly applied into NoCs.

3 Task Mapping Strategy

A scheme for removing a virtual-channel mechanism from dimension-ordered routers on tori is proposed. Our approach has the following designs: 1) each wrap-around channel has flexibility in that it can be dynamically enabled and disabled; 2) tasks are carefully assigned onto nodes so that as many wrap-around channels as possible are used without introducing deadlocks or performance degradation.

3.1 Reconfigurable Torus

We propose a variant of torus in which each wrap-around channel can be dynamically disabled. Each node in $k \times k$ 2-D torus is denoted as N_i, where $i = \{0, ..., k^2 - 1\}$. The set of unidirectional links along a given direction forms a unidirectional cycle. The unidirectional cycle that includes node N_i for $x+$ direction is denoted as ring R_i^{x+}. The other unidirectional cycles

for the $x-$, $y+$, and $y-$ directions are denoted as ring R_i^{x-}, R_i^{y+}, and R_i^{y-}, respectively. In Figure 2, ring R_0^{x+} is a unidirectional cycle $N_0 - N_1 - N_2 - N_3 - N_0$.

When all wrap-around channels are disabled, a deadlock-free path set is always guaranteed, but its performance is drastically decreased compared to that of a torus. Here, we employ a function to dynamically disable a part of the wrap-around channels in response to the communication pattern, so as to exploit most of the wrap-around channels. We call such a network a "reconfigurable torus." Note that the links are not electrically but logically disabled. That is, the routing table is just changed so as not to use the disabled link.

Figure 2 is an example of a 4×4 reconfigurable torus. Wrap-around channels for rings R_4^{x+}, R_4^{x-}, R_0^{y+}, R_0^{y-}, and R_8^{x-} are disabled in this example. Each router in reconfigurable tori has registers in which a set of configurations (enable or disable) of wrap-around channels for each direction is stored. When a wrap-around channel of a certain direction is disabled, the routing function becomes the same as that in a mesh network that requires no virtual channel for the dimension-order routing. The overhead area for a router in reconfigurable tori will be small, because only a 1-bit register is needed for each direction to switch its routing function. The synthesis result of a router for reconfigurable tori is shown in Section 4.1.

3.2 Mapping Algorithm

In order to assign tasks onto a reconfigurable torus, our task mapping strategy uses the communication pattern of the target application in which all the communication events are recorded. Our target application is stream processing as illustrated in Section 1. These applications are designed with system level description languages, and simulated in the early stage of design. We can obtain the total amount of communication data for each source-destination pair by analyzing the communication pattern. The total size of the communication data from task T_s to T_d is denoted as $D_{(s,d)}$. When $D_{(s,d)}$ is equal to *zero*, a communication path from task T_s to T_d is not used so it does not cause any deadlocks. Our task mapping strategy carefully assigns tasks onto a reconfigurable torus so that the valid communication paths ($D_{(s,d)} > 0$) do not form a cycle in each ring.

All possible mappings are represented in a tree structure as shown in Figure 3. Each mapping in this tree is represented as a linear list starting from a root node to the other nodes. In Figure 3, mapping $M = T_0 - T_1 - T_3 - T_2$ expresses that tasks T_0, T_1, T_3, and T_2 are placed onto nodes N_0, N_1, N_2, and N_3, respectively. Also, mapping $M' = T_0 - T_2$ expresses that tasks T_0 and T_2 are placed onto nodes N_0 and

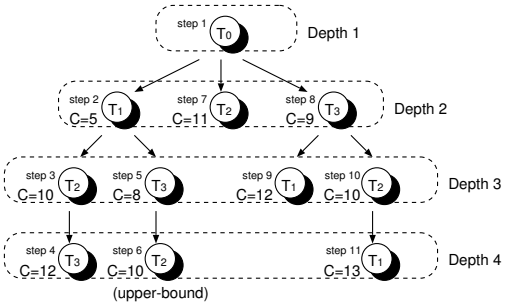

Figure 3: An example of searching tree of mapping. Four tasks are mapped onto four nodes.

N_1, but tasks T_1 and T_3 have not been yet placed. The latter mapping, which includes tasks not placed yet, is called "incomplete mapping."

Our task mapping strategy finds the best mapping that achieves deadlock-freedom and higher performance from all the possible mapping combinations. Since a search tree of all the possible combinations of mappings is huge, our mapping strategy employs the branch-and-bound method as a pruning technique to obtain a solution within a realistic time.

Now our mapping algorithm is briefly introduced. The *Search()* is a function that explores all the branches under a given mapping. In Figure 3, given that the current mapping is $T_0 - T_1$ (step 2), *Search()* explores $T_0 - T_1 - T_2$ and $T_0 - T_1 - T_3$ as branches under the current position. By recursively calling *Search()*, the proposed strategy walks through the whole searching space. For each mapping, deadlock-freedom is confirmed by *Cycle_Detect()* and the cost of the mapping is evaluated by *Cost()*. The cost in this strategy is an estimated average hop count. This strategy searches for deadlock-free mapping whose cost is the smallest. In Figure 3, the cost of mapping $T_0 - T_1 - T_3 - T_2$ (step 6) is ten. On the other hand, the cost of an incomplete mapping $T_0 - T_2$ (step 7) is eleven. Since the cost of all the branches under the mapping $T_0 - T_2$ will be higher than or equal to eleven, there is no chance of finding lower cost mapping than that of $T_0 - T_1 - T_3 - T_2$. By cutting such branches, computational efforts to find a minimal solution can be reduced.

The Task Mapping Algorithm:

1. (Initialize.) Variables are initialized. At step 1 in Figure 3, depth $d := 1$, mapping $M := T_0$, best mapping $M_{best} := \phi$, and upper-bound cost $C_{ub} := \infty$.
2. (Search down.) *Search(M,d)* is called in order to explore all branches under the mapping M. The *Search()* is stated below.
3. (Terminate.) When the upper-bound cost C_{ub} is less than ∞, the best mapping M_{best} is the solution. Otherwise, no valid solution exists.

Search(M,d): This function explores all branches under a given mapping M.

1. (Evaluate cost.) *Cost(M,d)* is called in order to evaluate the cost of mapping M. Notice that the *Cost()* is stated below.
   (a) When cost C is greater than or equal to upper-bound cost C_{ub}, the search of this branch is stopped and returned.
   (b) When cost C is less than upper-bound cost C_{ub} and mapping M is complete, it is the best mapping. Then the upper-bound cost is updated $C_{ub} := C$, and the best mapping is updated $M_{best} := M$.
   (c) When cost C is less than upper-bound cost C_{ub} and mapping M is incomplete, all branches under mapping M are going to be explored. For example, in step 2 in Figure 3, all branches under M are $M_0 = T_0 - T_1 - T_2$ and $M_1 = T_0 - T_1 - T_3$. Thus, $Search(M_0, d+1)$ and $Search(M_1, d+1)$ are called.

The *Cost(M,d)* is designed to satisfy that the cost of an incomplete mapping is always less than or equal to that of its child branches. Thus, the following relation must always be satisfied.

$$Cost(M, d) \leq Cost(M, d+1) \quad (1)$$

When the cost of an incomplete mapping is greater than upper-bound cost C_{ub}, it is worthless to continue exploring its child branches. In Figure 3, the searching tree is pruned at steps 7 and 9.

Cost(M, d): This function evaluates the cost of a given mapping M.

1. (Initialize a reconfigurable torus.) All wrap-around channels are set to "enable" in a reconfigurable torus T.
2. (Detect cycles.) The *Cycle_Detect(M,T)* stated below is called to check whether mapping M is cycle-free on reconfigurable torus T. When at least one cycle is detected, the wrap-around channels that cause cycles are disabled (reconfigurable torus T is updated). The *Cycle_Detect(M,T)* is then called again.
3. (Calculate cost.) The cost C of mapping M is calculated with the following equation:

$$C = \sum_{s=0}^{n-1} \sum_{d=0}^{n-1} H_{(s,d)} \times D_{(s,d)} \quad (2)$$

, where n is the number of nodes, $D_{(s,d)}$ is the total amount of communication data from task T_s to T_d, and $H_{(s,d)}$ is the path hop count from task T_s to T_d on reconfigurable torus T.

When both tasks T_s and T_d are fixed in mapping M, $H_{(s,d)}$ is simply calculated by dimension-order routing. On the other hand, when both tasks T_s and T_d are not fixed, the minimal hop count is estimated assuming that both tasks are placed on the nearest nodes where tasks are not yet placed. In the latter case, $H_{(s,d)}$ is often equal to one.

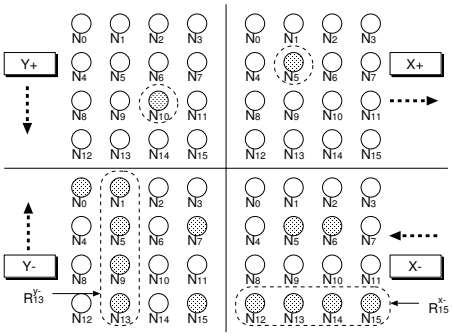

Figure 4: Bitmaps of each direction $(x+, x-, y+, y-)$.

Since the cost is calculated with Equation 2, pairs of tasks that transfer a large amount of data are placed so that their distances are the shortest. Using this cost function, tasks are placed so as to maximize the number of enabled wrap-around channels without causing performance degradation.

When both tasks T_s and T_d are not placed in mapping M, the minimal hop count is estimated assuming the best case. Thus, the cost of an incomplete mapping that includes tasks not yet placed is less than or equal to that of its child branches. This cost function is consistent with Relation 1.

Cycle_Detect(M,T): This confirms whether mapping M is cycle-free on reconfigurable torus T.

1. (Initialize bitmaps.) A bitmap of all nodes is allocated for each direction $(x+, x-, y+, y-)$ as shown in Figure 4.
2. (Route paths.) For each pair of source task T_s and destination task T_d:
   (a) When $D_{(s,d)}$ is equal to zero, or both T_s and T_d are not placed yet, return to 2.
   (b) A path from task T_s to T_d is calculated by dimension-order routing on reconfigurable torus T. The path is denoted as $P_{(s,d)}$. For example, $P_{(4,14)} = \{N_4, N_5, N_6, N_{10}, N_{14}\}$.
   (c) From path $P_{(s,d)}$, a source node, a destination node, and turning nodes that change the traveling direction are removed. For $P_{(4,14)}$, source node N_4, destination node N_{14}, and turning node N_6 are removed. Thus, $P'_{(4,14)} = \{N_5, N_{10}\}$.

(d) For each node in $P'_{(s,d)}$, the corresponding node is marked on the bitmap of its traveling direction. In the case of $P'_{(4,14)}$, node N_5 is marked on the $x+$ bitmap and N_6 is marked on the $y+$ bitmap (Figure 4).

3. (Detect cycles.) For each ring in reconfigurable torus T:

   (a) When all nodes on a ring are not marked, a cycle is not formed on the ring. We confirmed this with the following Theorem and Proof. In Figure 4, cycles are formed at ring R^{x-}_{15} and R^{y-}_{13}.

4. (Judge.) When no cycle is formed on all rings in reconfigurable torus T, mapping M is deadlock-free on reconfigurable torus T.

Theorem *When all nodes on a ring are not marked, a cycle is not formed on the ring.*

Proof *The Theorem noted above is proved with ring R^{x+}_i. It can also be proved with other rings in the same way. A source node, a destination node, and turning nodes are not marked in the bitmaps. Thus, when node N_i is marked on the $x+$ bitmap, there is at least one packet that occupies the $x-$ channel of N_i until the $x+$ channel of N_i is released. When all nodes on the ring are not marked, there is at least one channel in which no packet is waiting for its release. A cycle on the ring is broken by such channels. Thus, when all nodes on a ring are not marked, no cycle is formed on the ring.* ∎

3.3 The Case of Dynamic Traffic

It is possible that only a part of communication patterns cannot be known in the early stage of the system design. In such a case, the application contains unpredictable dynamic traffic that may form cyclic dependencies leading to deadlocks in our strategy.

We apply the simple concept to avoid deadlocks in the case of including the dynamic traffic as follows: an intermediate node temporarily stores packets to a certain destination node on its local memory, and reinjects them to their destination, as briefly introduced in Section 2[5]. Since the packet is once stored at an intermediate node, there is no cyclic dependency between before and after the intermediate node.

This concept introduces relatively large delay at this intermediate node in the area of NoCs that provide quite small delay on links and routers. Moreover, a large volume of dynamic traffic increases the interconnection latency and reduces the throughput, because an intermediate node cannot send or receive packets when its buffer is filled with dynamic packets waiting for their reinjection. In order to decrease the number of store-and-reinjetcted packets, we extend the proposed method for the case of mixing the static and dynamic traffic as follows:

1. The task mapping method shown in the previous subsection is applied only for the static traffic, which can be completely pre-analyzed. Cyclic channel dependencies on such static traffic can be cut by making at least one non-static-traffic channel on every ring.
2. Step (1) guarantees that only dynamic traffic will go through the non-static-traffic channel. Thus, in order to cut the cyclic channel dependency on the dynamic traffic, a node connected to the channel performs store-and-reinjection only for the dynamic traffic that will go through the channel.

All nodes are potentially expected to provide a buffer enough for packetization and de-packetization. Since this buffer can be used to temporarily storing the dynamic traffic, all nodes that have enough buffer can become an intermediate node without additional hardware resources.

4 Evaluation

4.1 Hardware Amount

We compare the hardware amount of routers for reconfigurable tori, conventional meshes and tori.

We implemented a five-port 32-bit wormhole router. In this design, the number of virtual channels is configurable among the two virtual channels (v2), three virtual channels (v3), and no virtual channel (v1). The router with virtual channels (v2 and v3) has four pipeline stages that consist of a routing computation, virtual-channel allocation, crossbar allocation, and crossbar traversal, while the v1 router does not have the virtual-channel allocation stage. Each pipeline stage has a buffer for storing a 1-flit.

We synthesized the following five router designs: **DOR+v1** is a router dedicated to dimension-order routing on 2-D meshes, so it does not have any virtual channels; **DOR+v1+R** is a router for dimension-order routing on 2-D reconfigurable tori; **CxN+v1** is a router that has routing tables and it does not have any virtual channels; **DOR+v2** is a router for dimension-order routing on 2-D tori, so two virtual channels are equipped; **CxN+v2** has routing tables with two virtual channels.

Figure 5 shows synthesis results of these five designs under a 0.18μm standard cell library. To clearly show the cost of a virtual-channel mechanism, we focus on DOR+v1 and DOR+v2. As shown in the graph,

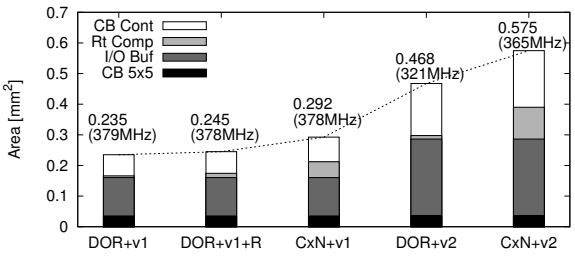

Figure 5: Hardware amount of v1 and v2 routers.

the DOR+v2 router consumes twice the larger area as that of DOR+v1. This is because the v2 routers have an additional pipeline stage for virtual-channel allocation, and they also need two sets of buffer for each virtual channel. Hence, the cost to newly add a virtual-channel mechanism is significant for the simple NoC routers that do not have virtual channels.

We shift to focus on the hardware amount of a reconfigurable-torus router. As shown in the graph, the size of DOR+v1+R router is decreased to 52.4% of DOR+v2 router. The main difference between DOR+v1 router and DOR+v1+R router is that each port in DOR+v1+R router is configurable in either torus mode (using a wrap-around channel) or mesh mode (not using a wrap-around channel). Although each port in DOR+v1+R router needs a 1-bit register to store this configuration, the additional cost is relatively small and in actuality, the total size of DOR+v1+R router increases only 4.3% when compared with DOR+v1 router.

4.2 Throughput

Throughput of the following three network architectures are compared: **Mesh+v1** is a 2-D mesh in which all routers do not require any virtual channels; **Torus+v2** is a 2-D torus and its routers have two virtual channels for deadlock-freedom; **RTorus+v1** is a 2-D reconfigurable torus without virtual channels.

4.2.1 Simulation Environments

A flit-level simulator written in C++ was developed for confirming deadlock-freedom and measuring throughput. We employ Mesh+v1, Torus+v2, and RTorus+v1 as network architectures. Every router has five ports; a port is connected to a single IP core, and the remaining ports are connected to neighboring routers. Wormhole switching is used in these routers. A simple model consisting of input buffers, a crossbar, and crossbar controllers is used for the switching fabric in the router. A header flit requires at least three clock cycles to be transferred to the next router or core; one cycle for the routing decision, one cycle for transferring a flit from an input channel to an output channel through a crossbar, and the remaining cycle for transferring the flit to the next router or host. We set packet length for 16 flits including one header flit.

We use NAS Parallel Benchmark (NPB) programs as the traffic patterns in this simulation. The NPB programs enable us to evaluate our mapping strategy with various traffic patterns on various sizes of NoCs. Also, their traffic includes similar access patterns (fork/join) to stream processing. We selected five matrix computation programs: BT, CG, IS, MG, and SP. The class of problem is "W", and the numbers of nodes for solving the problems are 9, 16, 32, 36, and 64. We use 17 application traces for this evaluation.

For RTorus+v1, the proposed mapping strategy is fully applied to place tasks without introducing deadlocks. For Mesh+v1 and Torus+v2, their task mappings are calculated so that pairs of tasks that transfer a large amount of data are placed near each other by using Equation 2. In this evaluation, mapping calculation time is equally limited to at most 60 minutes in every network for fair comparison.

In addition, our strategy is evaluated in the environments where 30% of packets are randomly generated as dynamic traffic. The results are denoted as **RTorus+v1+d**. RTorus+v1+d and RTorus+v1 are compared in the end of Section 4.2.2.

4.2.2 Simulation Results

Figure 6-9 show the accepted traffic versus the latency with BT traffic of 9-, 16-, 36-, and 64-node NoCs, respectively. The average hop counts on these mappings are also shown in the parenthesis. In the 9-node NoC case, the proposed mapper successfully assigns tasks onto a reconfigurable torus with all wrap-around channels enabled, and the throughput of RTorus+v1 is the same as that of Torus+v2. The same results are shown in 16- and 64-node cases. Whereas in the 36-node NoC, 12 of the 24 unidirectional wrap-around channels are prohibited in RTorus+v1, and its average hop count becomes longer than that of Torus+v2. Note that the throughput of RTorus+v1 still outperforms that of Mesh+v1. Since both SP and BT are based on similar algorithms, their communication patterns are alike, and the results of SP are similar to those of the BT program as shown in Figure 10-13.

Figure 20-22 show the results on IS traffic. The IS traffic is dominated by all-to-all communications. In the 64-node NoC case, all the wrap-around channels are disabled in RTorus+v1, and so the performance of RTorus+v1 is the same as that of Mesh+v1. In the 16-node case, all wrap-around channels can be exploited on RTorus+v1, even though all-to-all communication is used. This is because wrap-around channels can

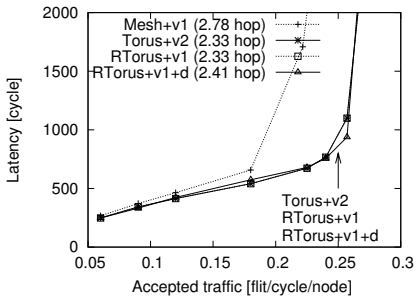

Figure 6: bt.w.9

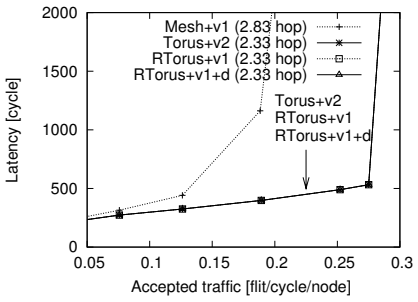

Figure 7: bt.w.16

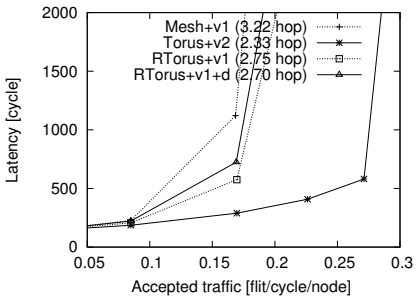

Figure 8: bt.w.36

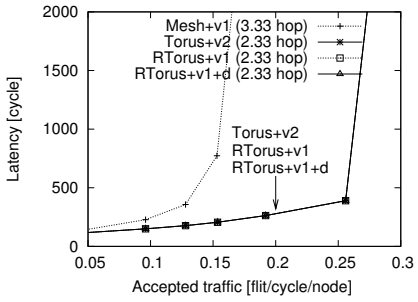

Figure 9: bt.w.64

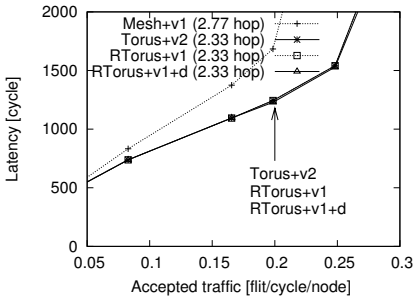

Figure 10: sp.w.9

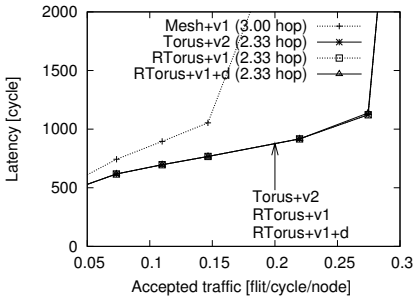

Figure 11: sp.w.16

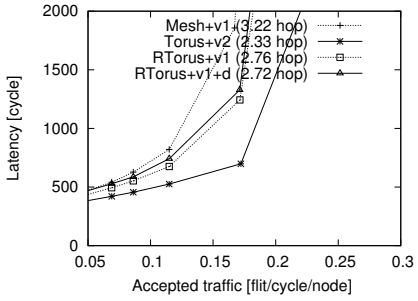

Figure 12: sp.w.36

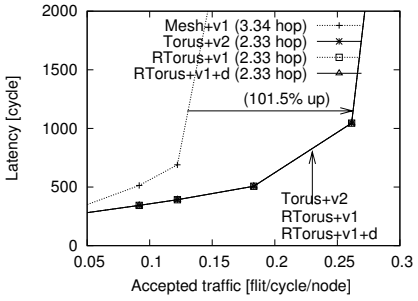

Figure 13: sp.w.64

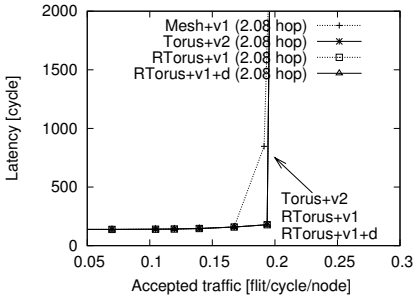

Figure 14: cg.w.16

be used only for one-hop-per-dimension packets in the case of 16-node tori, which introduce no deadlock.

The throughput on a 2-D reconfigurable-torus router achieves almost the same performance as that of a virtual-channel router on tori in ten of the 17 application traces. Target applications are often fixed in NoCs mostly for embedded applications. By carefully assigning tasks onto a reconfigurable torus using the proposed mapping strategy, the performance that goes near torus can be achieved with the cheap NoC routers in which all virtual-channel mechanism is omitted.

Finally, we show that our strategy can be used with unpredictable dynamic traffic. The differences between the results of RTorus+v1 and RTorus+v1+d (30% of packets are dynamic) vary depending on each application. For BT and SP traffic, the impact of the dynamic traffic is slight, because the number of packets reinjected at intermediate nodes is limited due to their small average hop counts. On the other hand, since the store-and-reinjection is frequently involved in IS programs, their performance is seriously reduced for the reason stated in Section 2. Although the task flow can be pre-analyzed in most of stream applications as illustrated in Figure 1, our strategy can be used for the case of mixing the static and dynamic traffic.

5 Conclusions

A scheme for removing a virtual-channel mechanism from dimension-ordered routers on tori is proposed by, 1) a reconfigurable torus in which each wrap-around channel can be dynamically enabled and disabled in each router, and 2) a task mapping strategy that analyzes the communication pattern and assigns tasks onto a reconfigurable torus so that as many wrap-around channels as possible are exploited without introducing deadlocks and performance degradation. Additionally, in the case of unpredictable dynamic traffic, we apply a simple mechanism to avoid their cyclic dependencies.

Through the area estimation of various routers, a

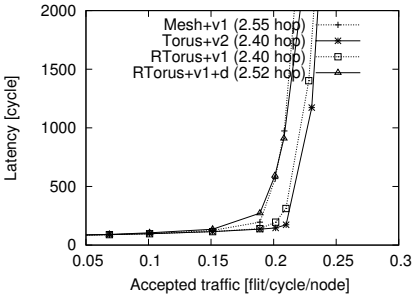

Figure 15: cg.w.32

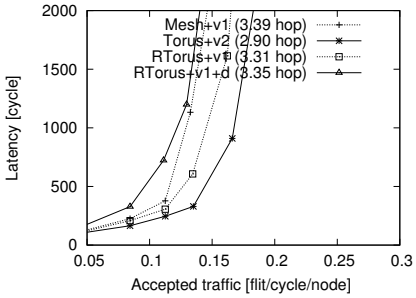

Figure 16: cg.w.64

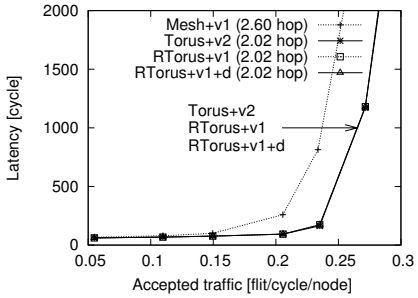

Figure 17: mg.w.16

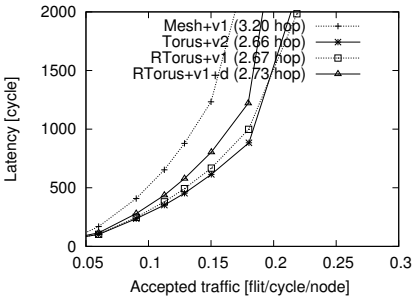

Figure 18: mg.w.32

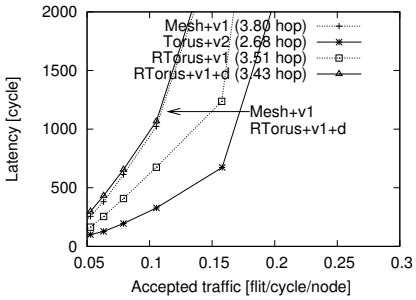

Figure 19: mg.w.64

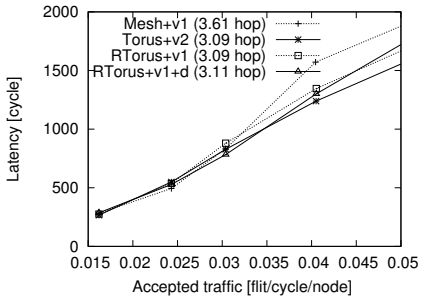

Figure 20: is.w.16

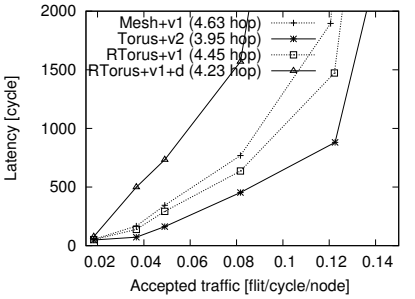

Figure 21: is.w.32

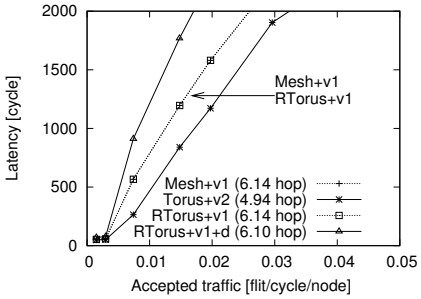

Figure 22: is.w.64

reconfigurable-torus router without any virtual channels can be implemented with a 52.4% area of a conventional router providing two virtual channels. On the other hand, a reconfigurable torus with the proposed task mapping strategy achieves almost the same performance as that on a virtual-channel router on a conventional torus in ten of the 17 application traces.

Acknowledgements

This work was supported by Joint Research Fund, "Network-on-Chip Architecture," National Institute of Informatics.

References

[1] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70–78, Jan. 2002.

[2] D. Burger and et. al. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[3] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Design Automation Conference*, pages 684–689, June 2001.

[4] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[5] J. Flich and et. al. Boosting the Performance of Myrinet Networks. *IEEE Transactions on Parallel and Distributed Systems*, 13(7):693–709, July 2002.

[6] J. Hu and R. Marculescu. Energy- and Performance-Aware Mapping for Regular NoC Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, Apr. 2005.

[7] T. Marescaux and et. al. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. In *Proceedings of the Field-Programmable Logic and Applications (FPL)*, pages 795–805, Sept. 2002.

[8] V. Puente and et. al. Adaptive Bubble Router: A Design to Improve Performance in Torus Networks. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 58–67, Sept. 1999.

[9] M. B. Taylor and et. al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22(2):25–35, Apr. 2002.